AD-A040 615 CALIFORNIA UNIV SANTA CRUZ F/G 17/2
RESPECIFYING THE TELEGRAM PROBLEM,(U)
FEB 77 W M MCKEEMAN N00014-76-C-0682
UNCLASSIFIED 77-2-001 NL

1 OF 1
ADA
040615

END
DATE
FILMED
7-77

12

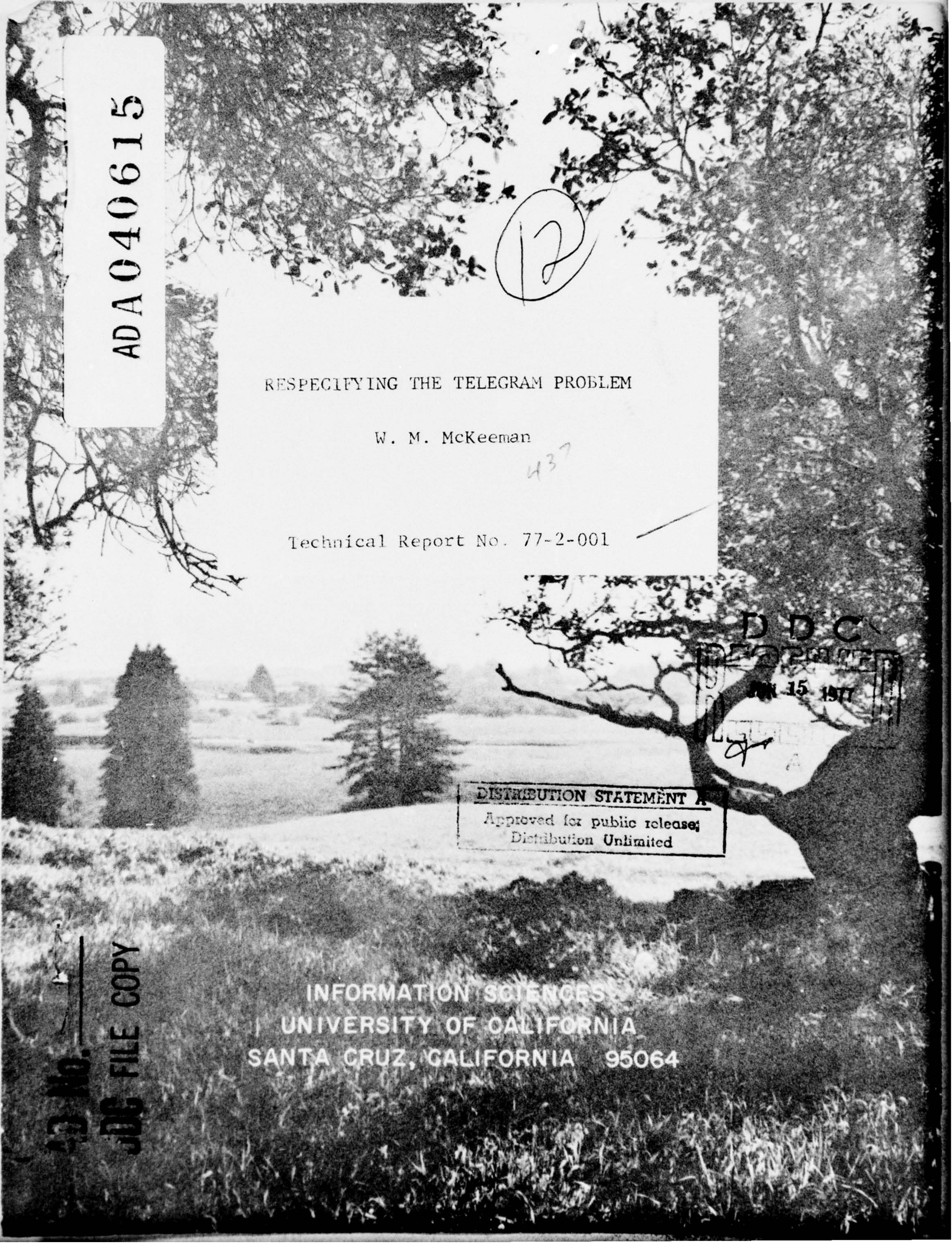
RESPECIFYING THE TELEGRAM PROBLEM

W. M. McKeeman

437

Technical Report No. 77-2-001

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064



(6) # RESPECIFYING THE TELEGRAM PROBLEM,

(10) W. M./McKeeman

Information Sciences
University of California
at
Santa Cruz, CA

(14) 77-2-ØØ1

388 073

February 2, 1977

(12) 16p.

(11) 2 Feb 77

ABSTRACT

is traced

The telegram problem was initially proposed as a programming exercise suitable for beginning students. A number of solutions have been published, all containing errors. In this paper we trace the trouble to the original specification and offer a new version. The most significant difference is the use of a grammatical notation for some parts of the specification. The new specification is then followed to a new solution. We conclude that the new specification is of higher quality, but not different in kind from the original.

The conclusion is

Key words and phrases:

Specification, grammars, structured programming

CR categories: 1.52, 4.2

(15)

This research was supported partially by the Office of Naval Research, Research Contract No. N00014-76-C-0682 and partially by the Computer Systems Research Group of the University of Toronto. The problem was suggested in the context of Working Group 2.3 of the International Federation of Information Processing Societies (Programming Methodology) and presented at the annual meeting at St. Pierre de Chartreuse, France, December 1976.

388Ø73

Introduction

Henderson specified and wrote a program to process telegrams [Henderson and Snowdon 72]. It has provoked a number of critiques [Henderson and Snowdon 72, Ledgard 74, Gerhart and Yelowitz 76, Zahn 76]. Much of the trouble arose from the lack of precision and completeness in the specification.

The original specification is given and discussed. Then a new specification is proposed and discussed. Finally a solution is presented.

Original Specification

The original specification was intended to be ready and understood by students in a programming course. It follows.

(1) A program is required to process a stream of telegrams. (2) This stream is available as a sequence of letters, digits and blanks (3) on some device (4) and can be transferred in sections of predetermined size into a buffer area where it is to be processed. (5) The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word "ZZZZ". The stream is terminated by the occurrence of the empty telegram, that is, a telegram with no words. (6) Each telegram is to be processed to determine the number of chargeable words (7) and to check for occurrences of overlength words. (8) The words "ZZZZ" and "STOP" are not chargeable (9) and words of more than twelve letters and considered overlength. (10) The result of the processing is to be a neat listing of the telegrams, (11) each accompanied by the word count (12) and a message indicating the

occurrence of an overlength word.

For purposes of discussion the specification is divided into 12 parts (each starting at the corresponding number). Parts 2 and 5 describe the form of the input. They are summarized in the following grammar (the notation is described in Appendix I).

$$\boxed{\begin{array}{l}
\text{input = blank* stream blank*} \\
\text{stream = (telegram blank}^{+}\text{)* 'ZZZZ'} \\
\text{telegram = (word blank}^{+}\text{)}^{+}\text{ 'ZZZZ'} \\
\text{word = (letter}|\text{digit)}^{+}\text{ - 'ZZZZ'}
\end{array}}$$

Part 4 implies that the length of the input is a multiple of the size of the buffer. It is, however, unreasonable to expect the user to prepare input conforming to that constraint, or even with enough "padding" to ensure that the last buffer can be filled. This raises a serious difficulty. The processing that detects the final 'ZZZZ' is done _after_ the buffer that contains it is filled. Thus the end-of-file condition on the device in part 3 may occur before a transfer from the device to the buffer is complete. Implementation cannot proceed until this difficulty is resolved.

We therefore effectively relieve the programmer of responsibility for this by _specifying_ that the transfer operation deliver the input in multiples of the buffer size until there is less than a buffer-full of characters remaining. One final transfer then is allowed which delivers all the remaining input (if any) and pads the buffer out with blanks. Any further requests for input from the program are in error.

This incidentally ensures that there is at least one blank after the final 'ZZZZ' (necessary for otherwise it would not be a word, hence not the final 'ZZZZ').

All of the foregoing can be summarized in the grammar below, where M is the size of the buffer.

input = stream_structure & buffer_structure
stream_structure = blank* stream blank^{+}
stream = (telegram blank^{+})* 'ZZZZ'
telegram = (word blank^{+})$^{+}$ 'ZZZZ'
word = (letter|digit)$^{+}$ - 'ZZZZ'
buffer_structure = buffer^{+}
buffer = character^{M}
character = letter|digit|blank

Note that a word in the input stream may well break across a buffer boundary yet must remain intact during processing. Henderson's program in fact makes two words out of each such broken word. Brian Randell pointed out to me that this is a case of structure clash [Jackson 76] and in fact the appearance of the connective "&" in the grammar above advertises the possibility.

The required processing involves some detail on the words themselves (parts 6,7,8,9), as well as the production of output (parts 10,11,12). The latter is specified by the single adjective "neat", causing one to wonder what can be meant by a "correct" program [Ledgard 74, Gerhart & Yelowitz 76]. Zahn's challenge to find yet one more error [76] seems beside the point in the face of this.

We therefore proceed to a new specification, using a grammar to expose the necessary detail.

New Specification

Write a program to process a stream of telegrams. The input and output are partially described by the grammar below

input = (blank* stream blank$^+$) & input_buffer$^+$

stream = (input_telegram blank$^+$)* 'ZZZZ'

input_telegram = (useable_word blank$^+$)$^+$ 'ZZZZ'

useable_word = chargeable_word| 'STOP'

chargeable_word = word - 'STOP' - 'ZZZZ'

word = reasonable_word|overlength_word

reasonable_word = non_blank$_1^{12}$

overlength_word = truncated_word overflow

truncated_word = non_blank^{12}

overflow = non_blank$^+$

output = (output_telegram analysis)*

output_telegram = ((output_line pad) & output_buffer)$^+$

output_line = output_word (blank output_word)*

output_word = (reasonable_word|truncated_word) - 'ZZZZ'

analysis = word_count overlength_warning$^?$

word_count = (digit$^+$ blank 'WORDS' pad) & output_buffer

overlength_warning = ('LONG' blank 'WORD' pad) & output_buffer

pad = blank*

input_buffer = character^M

output_buffer = character^N

character = non_blank|blank

non_blank = letter|digit

The input may be transferred in sections of M characters from the input device to the input buffer. If there are less than M characters of input remaining, they are extended with sufficient blanks to fill the buffer, the transfer is allowed, and no further input may be requested.

Each input telegram consists of a sequence of useable words. The corresponding output telegram consists of the same sequence in which each overlength_word is replaced by its truncated_word part. Overflows, extra blanks, and the terminators 'ZZZZ' are thereby discarded.

The number of chargeable words in each input telegram is to be determined and output according to the format specified by the phrase "word_count". Furthermore, if an input telegram contains an overlength word, the optional warning is to be output.

The output may be transferred from the output buffer to the output device in sections of N characters. The length of the pad on the end of each output line is to be minimized.

The following restrictions are to be assumed:

$0 < M \le 1024$

$12 \le N \le 1024$

at most 999999 chargeable words per telegram

Tricky Tests

The following section contains a list of tests designed to illuminate some of the problems.

(1) ZZZZ itself (null stream)

(2) any word across an input buffer boundary

(3-7) STOP or ZZZZ at the end of the input buffer, with the next character blank or non-blank

(8) any word across an output buffer boundary

(9) any word at end of the output buffer

(10) a telegram with leading blanks

(11) a non-null telegram with no chargeable words

(12) a telegram for which some output line does not end in a blank (watch for leading blanks, or wholly blank lines)

(13) M=1

(14) N=12

(15)

```
HELLO MOTHER STOP
 2 WORDS LONGWORD
HERENOW 2 WORDS
LONG WORD       Z
ZZZ             Z
ZZZZ            Z
ZZZ ZZZZ
```

with N=M=17

(16) a stream with a multiple of M characters of input ending in 'ZZZZ' (i.e., not having a trailing blank).

Refining the specification

The grammars as they stand separate the problem into three:

(1) parsing the input

(2) generating the output

(3) establishing the correspondence between them

It is helpful to use the same linguistic tools for one more level of refinement before starting the actual implementation. The approach is similar to that of Jackson [76].

I propose a feedback-free sequence of conceptual modules, each communicating to the next via an intermediate language [McKeeman & DeRemer 74]. From the module viewpoint the program has the form

program = module$^+$

module = input output

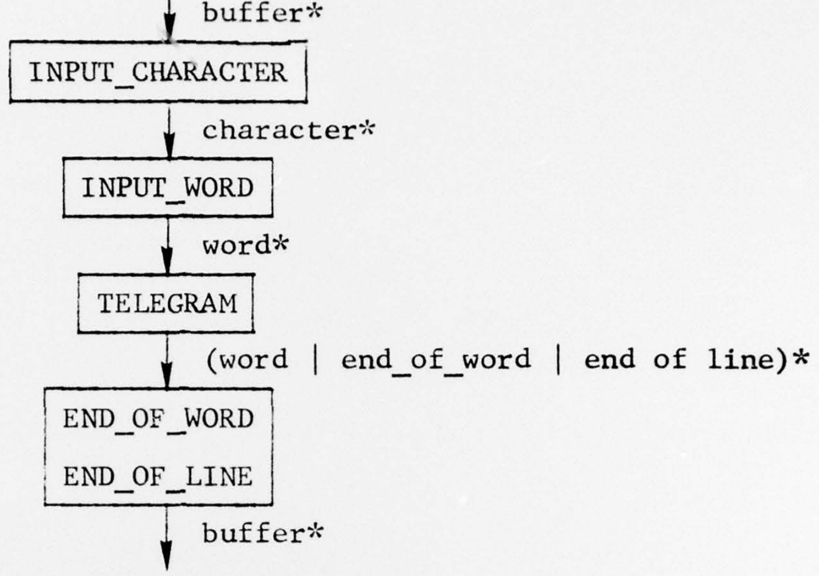
The function of each module is to produce the output from the input. From the intermediate language viewpoint the program has the form

program = input intermediate* output

intermediate = output & input

The intersection (&) of the output actually produced by a module, and the input actually acceptable to its successor is the range over which the connected modules can communicate.

For this problem it turns out that four modules provide enough structure to resolve the clashes. Using the procedure names given in the following section, the overall form of the program and its data flow is as follows.



The corresponding grammars are given below.

input = input_buffer
input_buffer = character^{M}
output = character*
character = letter | digit | blank

Module INPUT_CHARACTER

input = blank* stream blank^{+}
stream = word (blank^{+} word)*
word = reasonable_word | overlength_word
reasonable_word = non_blank$_{1}^{12}$
overlength_word = truncated_word overflow
truncated_word = non_blank^{12}
overflow = non_blank^{+}
output = (reasonable_word | truncated_word)*
non_blank = letter | digit

Module INPUT_WORD

input = input_telegram* 'ZZZZ'

input_telegram = useable_word$^+$ 'ZZZZ'

useable_word = chargeable_word | 'STOP'

chargeable_word = input_word-'STOP'-'ZZZZ'

input_word = non_blank$_1^{12}$

output = (output_telegram analysis)*

output_telegram = output_word$^+$ end_of_line

output_word = useable_word end_of_word

analysis = word_count overlength_warning$^?$

word_count = digit$^+$ end_of_word 'WORDS' end_of_word end_of_line

overlength_warning = 'LONG' end_of_word 'WORD' end_of_word end_of_line

non_blank = letter | digit

Module TELEGRAM

Note that the symbols end_of_word and end_of_line are not defined. They stand for unique terminal symbols.

input = (word | end_of_word | end_of_line)*

word = non_blank$_1^{12}$

output = output_buffer*

output_buffer = (non_blank | blank)^N

non_blank = letter | digit

Module END_OF_WORD & END_OF_LINE

An implementation

The program was implemented in SP/k, a subset of PL/I [Hume & Holt 75]. The correspondence between the new specification and the refined specification, and the correspondence between the refined specification and the program is not formally established, but rather intuitive in nature. It is given in appendix II.

Conclusions

The grammar provided an enhanced specification but still left some detail to descriptive text. Errors showed up during implementation. Thus it would seem that we offer a technique of higher quality but not different in kind from the original.

References

1. Gerhart & Yelowitz, Observations of fallibility in applications of modern programming methodologies, IEEE TSE SE2-3 (9/76) pp. 195-207.
2. Henderson & Snowdon, An Experiment in Structured Programming, BIT 12 (1972) 38-53.
3. Hume & Holt, Structured Programming Using PL/1 and SP/k, Reston Publishing Co. (1975).
4. Jackson, Constructive Methods of Program Design, submitted to ECI, Amsterdam (8/76).
5. Ledgard, The Case for Structured Programming, BIT 14 (1974) 45-57 (N.B., reference printed in BIT is wrong!).
6. McKeeman & DeRemer, Feedback-free modularization of compilers, Proc. third Conference on Programming Languages, Kiel (March 1974).
7. Zahn, letter to Balzer of 76.7.9.

Acknowledgments

The author is indebted to Jim Horning, Brian Randell, Sharon Sickel and Ric Holt for many helpful discussions and patient readings of earlier drafts.

Appendix I

The grammar used in this paper are a variant of context-free grammars. Terminal symbols are delimited by apostrophes. Non-terminal symbols are denoted by identifiers. Three of them have standard definitions as given below.

letter = 'a'|'b'|'c'...|'z'|'A'|'B'|...|'Z'

digit = '0'|'1'|'2'|... '9'

blank = ' '

All others appear on the left-hand-side of some rule in the grammar.

The metasymbol "|" significes disjunction of phrase classes. That is, for phrase classes K and L, K|L means those strings that lie in either class K or class L or both. "&" signifies conjunction. For phrase classes K and L, K&L means those strings that lie in both class K and class L. "-" significes set difference. For phrase classes K and L, K-L means all those strings that lie in K except for those that lie in L. The exponents are defined by the set of relations given below. If L is any phrase class:

L^0 = null string

$L^n = L\ L^{n-1}$ for $0<n$ (i.e. exactly n L's)

$L_n^n = L^n$

$L_m^n = L^n | L_m^{n-1}$ for $0 \leq m < n$ (i.e. m to n L's)

$L* = L_0^\infty$ (i.e. 0 or more L's)

$L^+ = L_1^\infty$ (i.e. 1 or more L's)

$L^? = L_0^1$ (i.e. optional L)

```
/*                         APPENDIX II                        */
/*N.B.  SP/K HAS NO FACILITY TO RESPOND TO END-OF-FILE */
STREAM:PROCEDURE OPTIONS(MAIN):
   /* INTERMODULAR COMMUNICATIONS */
   /*                               M                   */
   DECLARE INPUT BUFFER CHARACTER(17) VARYING;
   DECLARE CHR CHARACTER(1) VARYING:
   DECLARE WORD CHARACTER(12) VARYING:
   DECLARE OUTPUT_WORD CHARACTER(12)VARYING;
   /*                                N                  */
   DECLARE OUTPUT_BUFFER CHARACTER(17) VARYING;
   DECLARE OVERLENGTH BIT(1);

   INPUT_CHARACTER:PROCEDURE;
      IF LENGTH(INPUT_BUFFER)=0 THEN
         /*                             M           */
         GET SKIP EDIT(INPUT_BUFFER)(A(17)):
      CHR=SUBSTR(INPUT BUFFER,1,1);
      INPUT_BUFFER=SUBSTR(INPUT_BUFFER,2);
      END INPUT_CHARACTER;

   INPUT_WORD:PROCEDURE;
     DO WHILE(CHR= ' ');
        CALL INPUT_CHARACTER;
        END;

     WORD='';
     DO WHILE(CHR¬= ' ');
        IF LENGTH(WORD)<12 THEN
           WORD=WORD||CHR;
        ELSE OVERLENGTH = '1'B;
        CALL INPUT_CHARACTER;
        END;
     END INPUT_WORD

   END OF WORD:PROCEDURE;                          /*     N  */
      IF LENGTH(OUTPUT_WORD)+LENGTH(OUTPUT_BUFFER)>17 THEN
         DO;
            PUT SKIP LIST(OUTPUT_BUFFER);
            OUTPUT_BUFFER='';
            END;
      IF LENGTH(OUTPUT_BUFFER)¬=0 THEN
         OUTPUT_BUFFER=OUTPUT_BUFFER||' ';
      OUTPUT_BUFFER=OUTPUT_BUFFER||OUTPUT_WORD;
      END END_OF_WORD;

   END_OF_LINE:PROCEDURE;
      PUT SKIP LIST(OUTPUT_BUFFER);
      OUTPUT_BUFFER='';
      END END_LINE
```

```
NUMERIC_TO_STRING:PROCEDURE(NUMERIC)RETURNS(CHARACTER(6)
   VARYING);
   DECLARE NUMERIC FIXED;
   DECLARE STRING CHARACTER(6)VARYING;
   IF NUMERIC=0 THEN
      RETURN('0');
   STRING='';
   DO WHILE(NUMERIC>0);
      STRING=SUBSTR('0123456789',MOD(NUMERIC,10)+1,1)||STRING;
      NUMERIC=NUMERIC/10;
      END;
   RETURN(STRING);
   END NUMERIC TO_STRING;

TELEGRAM:PROCEDURE;
   DECLARE COUNT FIXED;

   COUNT=0;
   OVERLENGTH='0'B;
   DO WHILE(WORD¬='ZZZZ');
      OUTPUT_WORD=WORD;
      CALL END_OF_WORD;
      IF WORD¬='STOP' THEN
         COUNT=COUNT+1;
      CALL INPUT_WORD;
      END;

   CALL END_OF_LINE;
   OUTPUT_WORD=NUMERIC_TO_STRING(COUNT);
   CALL END_OF_WORD;
   OUTPUT_WORD='WORDS';
   CALL END_OF_WORD;
   CALL END_OF_LINE;

   IF OVERLENGTH THEN
      DO;
         OUTPUT_WORD='LONG';
         CALL END_OF_WORD;
         OUTPUT_WORD='WORD';
         CALL END_OF_WORD;
         CALL END_OF_LINE;
         END;
   END TELEGRAM:

INPUT_BUFFER='';
OUTPUT_BUFFER='';
CALL INPUT_CHARACTER;
DO WHILE('1'B);
   CALL INPUT_WORD;
```

```
          IF WORD='ZZZZ' THEN
             RETURN;
          CALL TELEGRAM;
          END;
       END STREAM;


END OF COMPILATION.  942 BYTES OF CODE GENERATED.

EXECUTION BEGINS.

ABCDEFGHIJKL
1 WORDS
LONG WORD

END OF EXECUTION.  468 BYTES OF DATA STORAGE USED.
   376 STATEMENTS EXECUTED.  3 LINES OF OUTPUT.
```